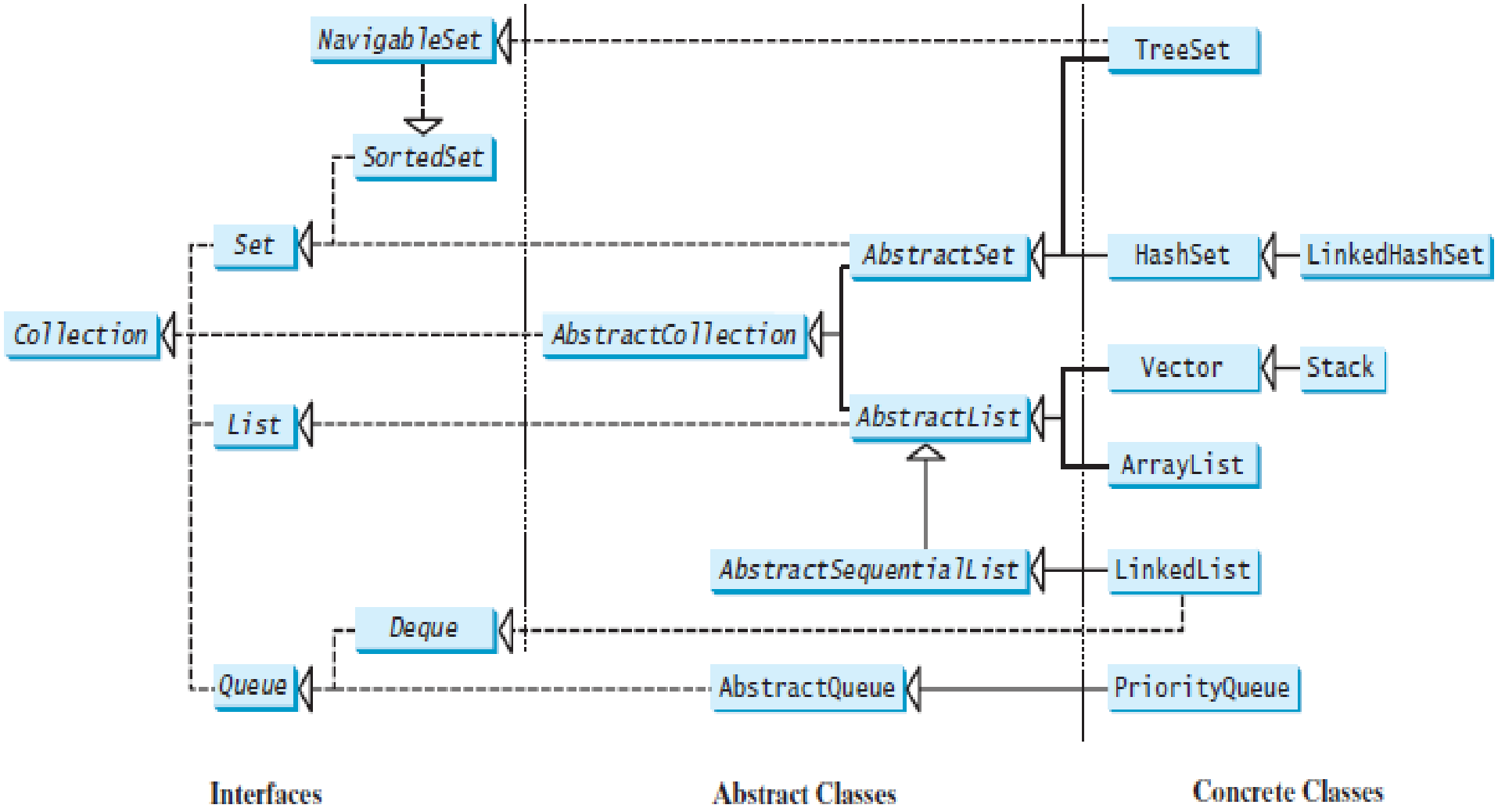
Programming in Java

Java Collection Framework

Introduction

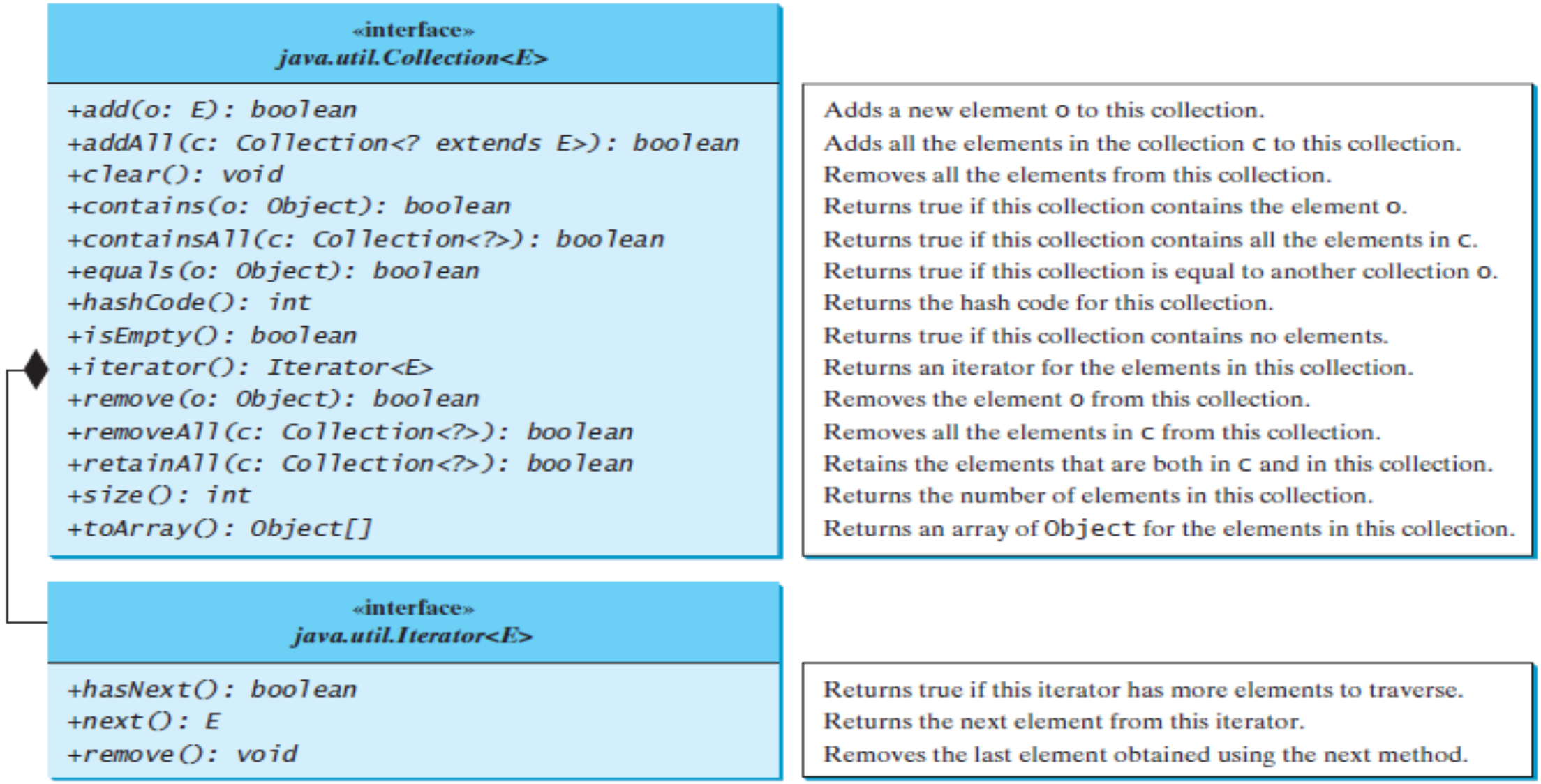
- Java Collection Framework
- Collection/Container Object/Collection Object
- We will be creating objects of appropriate class and using their in built functions.
- TestArrayList.java[DemoProgram]
- Package is java.util
- Main Types of collections are set, lists and maps.

Collection



All the concrete classes in the Java Collections Framework implement the java.lang. Cloneable and java.io. Serializable interfaces. Thus their instances can be cloned and serialized.

Collection Interface and AbstractCollection class



Collection Interface and AbstractCollection class

- Set Operations: The methods **addAll**, **removeAll**, and **retainAll** are similar to the set union, difference, and intersection operations.
- Query Operations: size, contains, containsAll, isEmpty.
- **toArray()** method that returns an array representation for the collection.
- Iterator
- Unsupported Operations
- **public void** someMethod()

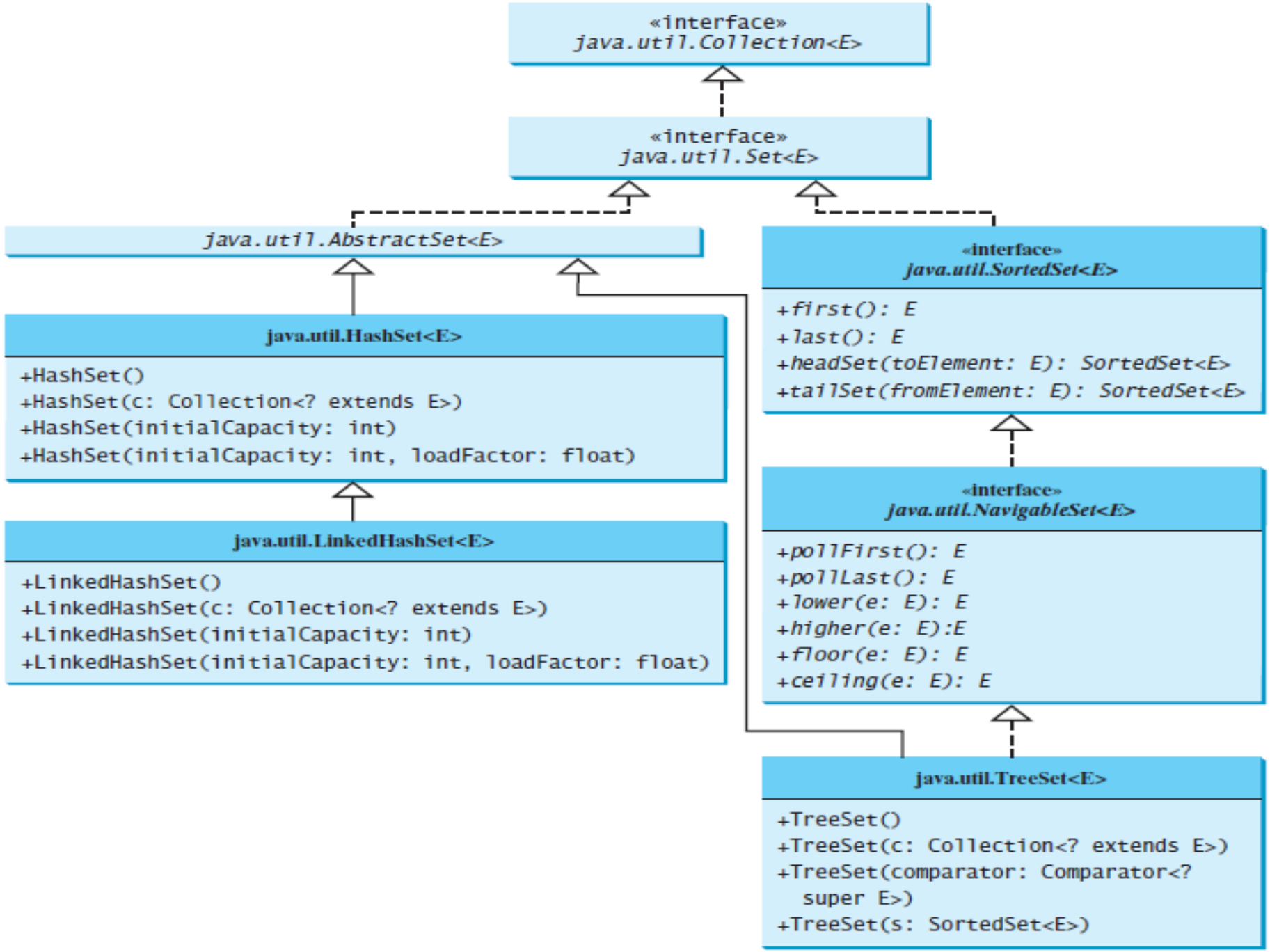
```
{
    throw new UnsupportedOperationException("Method not supported");
}
```

sets

- The **AbstractCollection** class is a convenience class that provides partial implementation for the **Collection** interface. It implements all the methods in **Collection** except the **size** and **iterator** methods.

- The **AbstractSet** class is a convenience class that extends **AbstractCollection** and implements **Set**.
- The **AbstractSet** class provides concrete implementations for the **equals** method and the **hashCode** method.

- Since the **size** method and **iterator** method are not implemented in the **AbstractSet** class, **AbstractSet** is an abstract class.


«interface»
java.util.Collection<E>
«interface»
java.util.Set<E>
java.util.AbstractSet<E>
«interface»
java.util.SortedSet<E>
+first(): E
+last(): E
+headSet(toElement: E): SortedSet<E>
+tailSet(fromElement: E): SortedSet<E>
java.util.HashSet<E>
+HashSet()
+HashSet(c: Collection<? extends E>)
+HashSet(initialCapacity: int)
+HashSet(initialCapacity: int, loadFactor: float)
«interface»
java.util.NavigableSet<E>
+pollFirst(): E
+pollLast(): E
+lower(e: E): E
+higher(e: E):E
+floor(e: E): E
+ceiling(e: E): E
java.util.LinkedHashSet<E>
+LinkedHashSet()
+LinkedHashSet(c: Collection<? extends E>)
+LinkedHashSet(initialCapacity: int)
+LinkedHashSet(initialCapacity: int, loadFactor: float)
java.util.TreeSet<E>
+TreeSet()
+TreeSet(c: Collection<? extends E>)
+TreeSet(comparator: Comparator<?
super E>)
+TreeSet(s: SortedSet<E>)

TreeSet

- TreeSet provides an implementation of the Set interface that uses a tree for storage. Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly.
- The TreeSet class supports four constructors.

Methods from Navigable Interface

- lower(E e): Returns the greatest element in this set strictly less than the given element, or null if there is no such element.
- higher(E e): Returns the least element in this set strictly greater than the given element, or null if there is no such element.
- floor(E e): Returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
- ceiling(E e): Returns the least element in this set greater than or equal to the given element, or null if there is no such element.
- pollFirst(): Retrieves and removes the first (lowest) element, or returns null if this set is empty.
- pollLast(): Retrieves and removes the last (highest) element, or returns null if this set is empty.

TreeSet

- **SortedSet** is a subinterface of **Set**, which guarantees that the elements in the set are sorted.
- Additionally, it provides the methods **first()** and **last()** for returning the first and last elements in the set.
- **headSet(toElement)** and **tailSet(fromElement)** for returning a portion of the set whose elements are less than **toElement** and greater than or equal to **fromElement** respectively.
- Natural Order and Order by comparator
- TestTreeSet.java[Demo Program]

Comparator Interface

- When we want to insert elements into a tree set and elements are not be instances of java.lang.Comparable.
- We can define a comparator to compare these elements. To do so, create a class that implements the java.util.Comparator interface.
- The Comparator interface has two methods, compare and equals.
- public int compare(Object element1, Object element2)

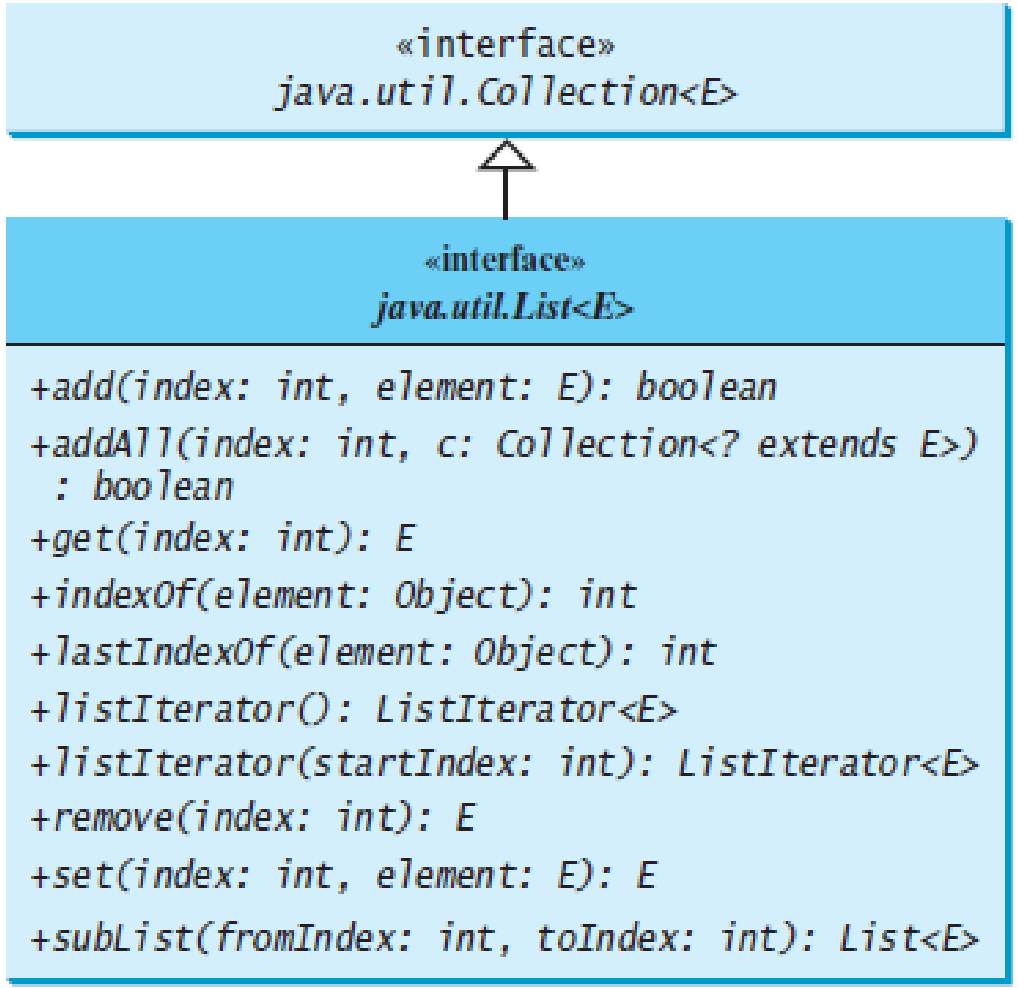
  Returns a negative value if element1 is less than element2, a positive value if element1 is greater than element2, and zero if they are equal.
- public boolean equals(Object element)

  Returns true if the specified object is also a comparator and imposes the same ordering as this comparator.
- TestTreeSetWithComparator.java[Demo. Program]

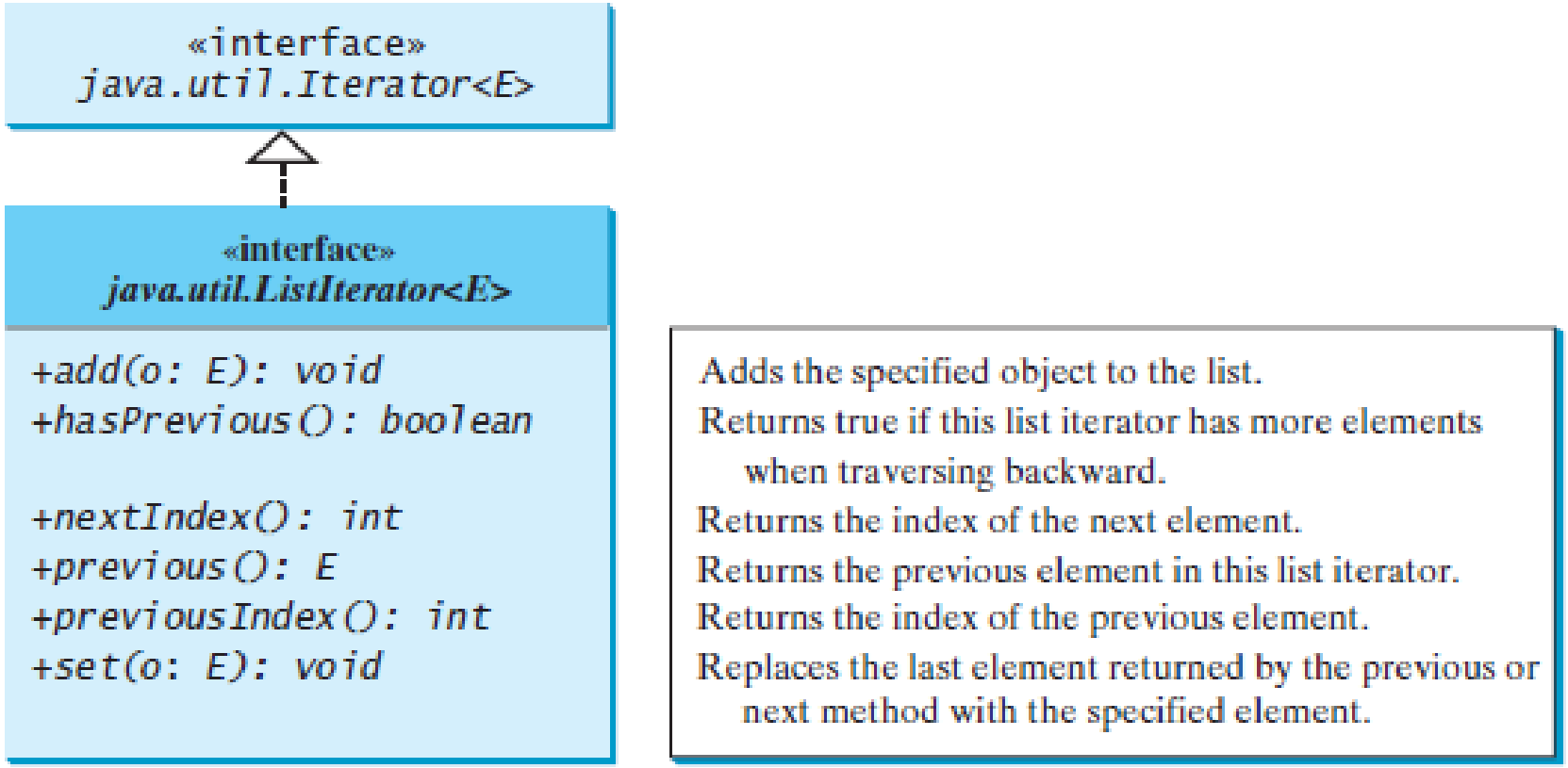
Lists

- To allow duplicate elements to be stored in a collection, we need to use a list.
- A list can not only store duplicate elements but also allow the user to specify where they are stored. The user can access elements by an index.
- The **List** interface extends **Collection** to define an ordered collection with duplicates allowed.
- The **List** interface adds position-oriented operations, as well as a new list iterator that enables the user to traverse the list bidirectionally.

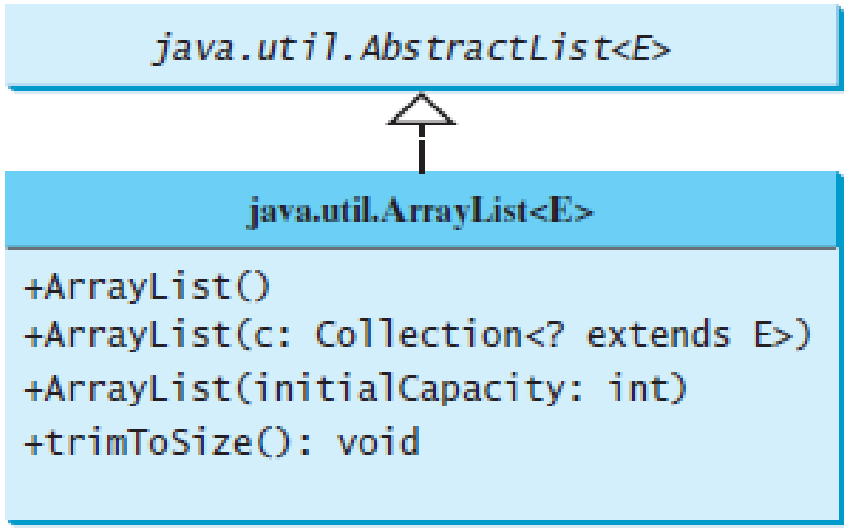
List


«interface»
java.util.Collection<E>
«interface»
java.util.List<E>
+add(index: int, element: E): boolean
+addAll(index: int, c: Collection<? extends E>) : boolean
+get(index: int): E
+indexOf(element: Object): int
+lastIndexOf(element: Object): int
+listIterator(): ListIterator<E>
+listIterator(startIndex: int): ListIterator<E>
+remove(index: int): E
+set(index: int, element: E): E
+subList(fromIndex: int, toIndex: int): List<E>
Adds a new element at the specified index.
Adds all the elements in c to this list at the specified index.
Returns the element in this list at the specified index.
Returns the index of the first matching element.
Returns the index of the last matching element.
Returns the list iterator for the elements in this list.
Returns the iterator for the elements from startIndex.
Removes the element at the specified index.
Sets the element at the specified index.
Returns a sublist from fromIndex to toIndex-1.

java.util.ListIterator<E>

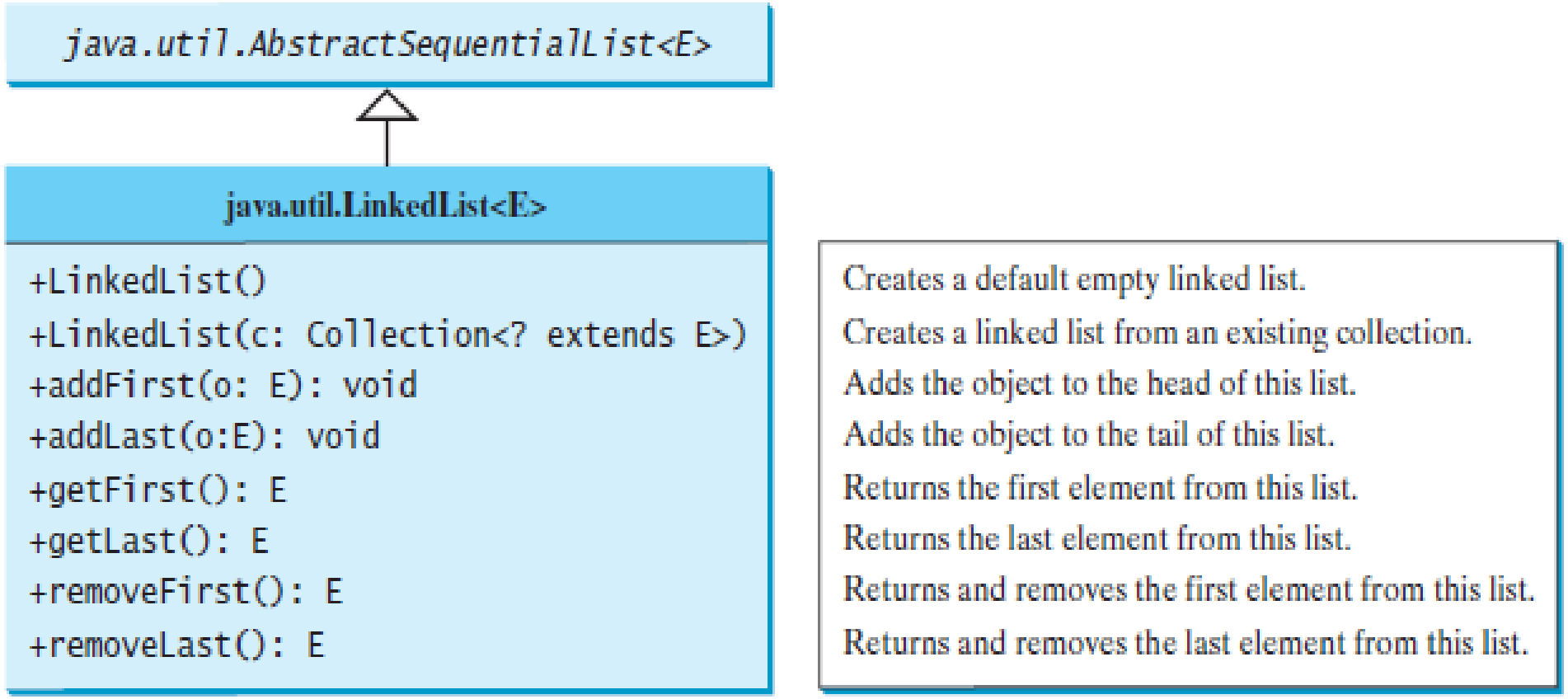


Arrays Lists



Creates an empty list with the default initial capacity.	
Creates an array list from an existing collection.	
Creates an empty list with the specified initial capacity.	
Trims the capacity of this `ArrayList` instance to be the list's current size.	

LinkedLists

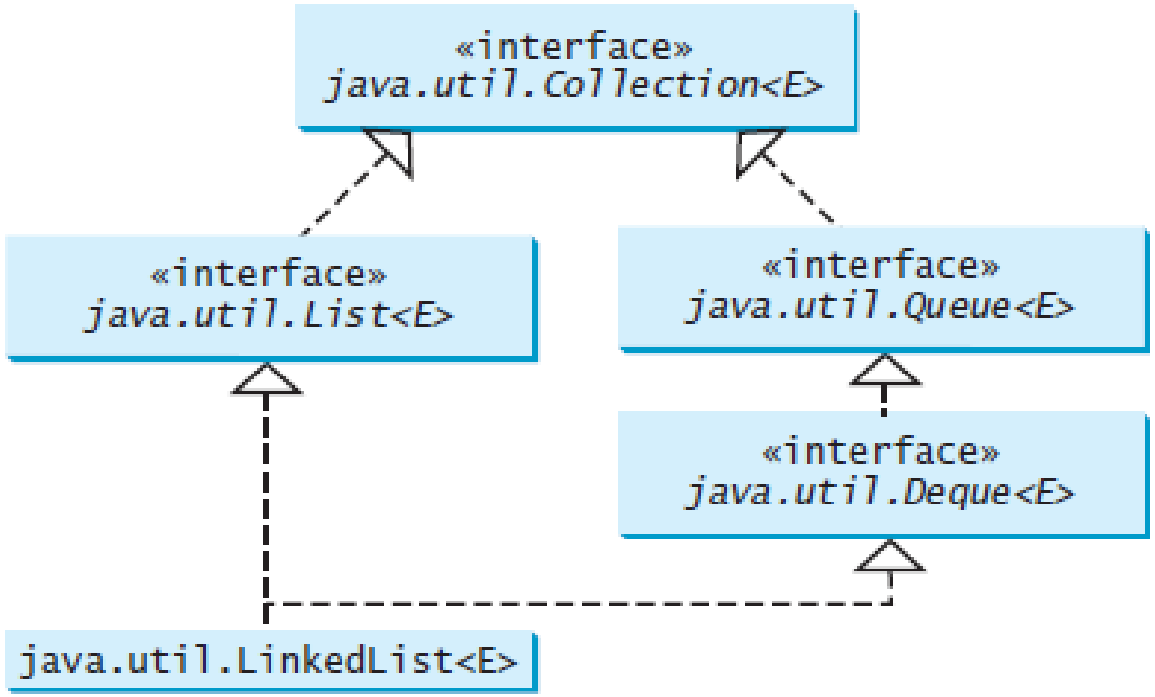


TestArrayList.java

TestArrayAndLinkedList.java[Demo. Program]

Deque and LinkedList

- The LinkedList class implements the Deque interface, which extends the Queue interface. So we can use LinkedList to create a queue.



Deque

- Usually pronounced as deck, a deque is a double-ended-queue. A double-ended-queue is a linear collection of elements that supports the insertion and removal of elements at both end points.
- The Deque interface is a richer abstract data type than both Stack and Queue because it implements both stacks and queues at the same time.
- The Deque interface, defines methods to access the elements at both ends of the Deque instance. Methods are provided to insert, remove, and examine the elements.
- Predefined classes like ArrayDeque and LinkedList implement the Deque interface.

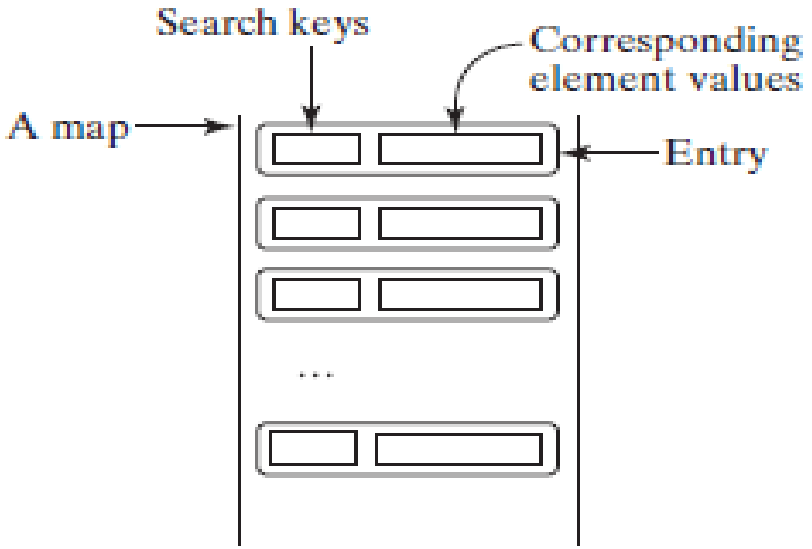
LinkedList methods and respective Queue methods

Type of Operation	First Element (Beginning of the Deque instance)	Last Element (End of the Deque instance)
Insert	addFirst(e) offerFirst(e)	addLast(e) offerLast(e)
Remove	removeFirst() pollFirst()	removeLast() pollLast()
Examine	getFirst() peekFirst()	getLast() peekLast()

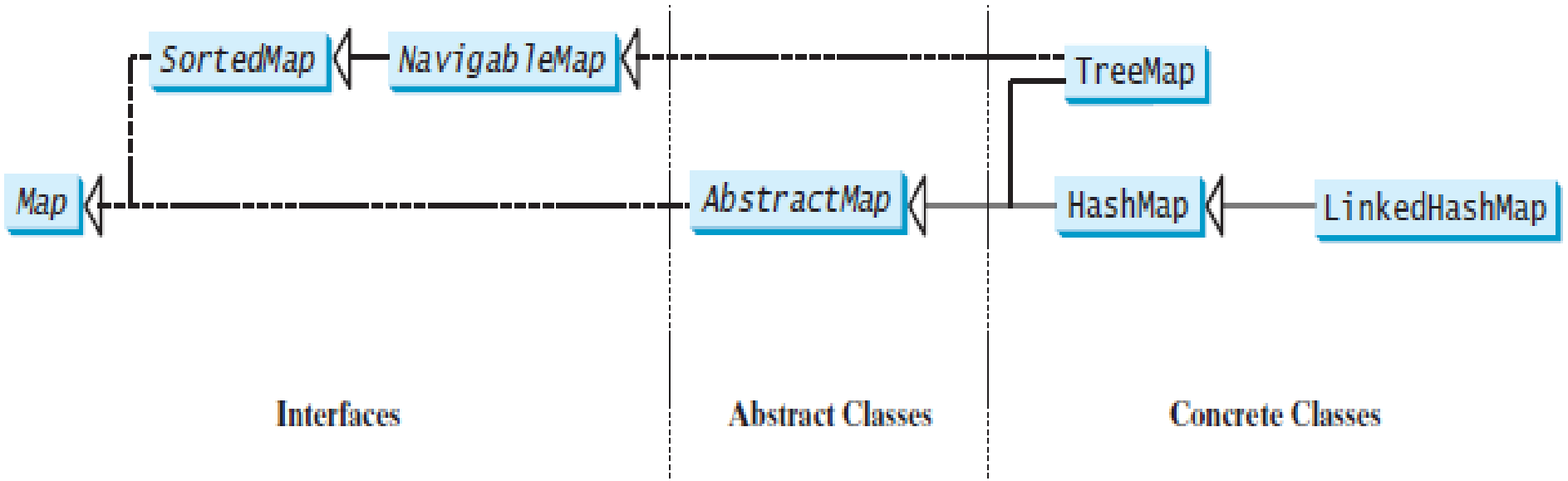
TestDeque.java[Demo Program]

Maps

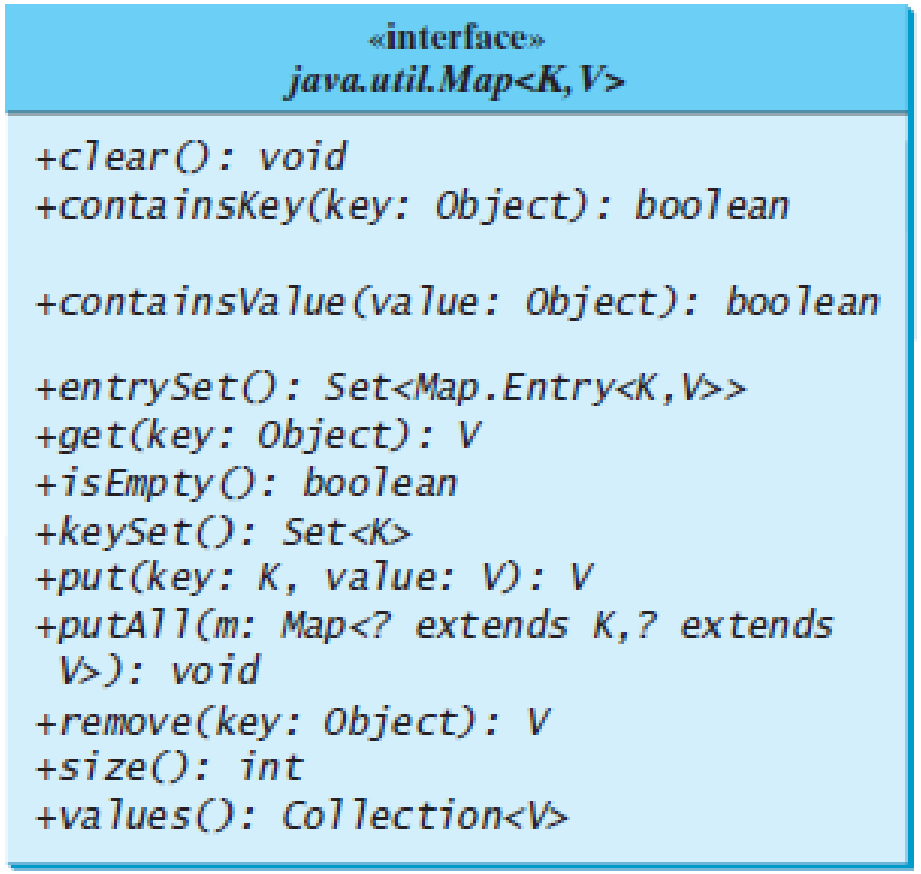
- Optimizes Searching.
- A map is a container that stores the elements along with the keys.
- The keys are like indexes.
- In Map, the keys can be any objects. A map cannot contain duplicate keys. Each key maps to one value. A key and its corresponding value form an entry, which is actually stored in a map.



Map interface



java.util.Map<K,V>



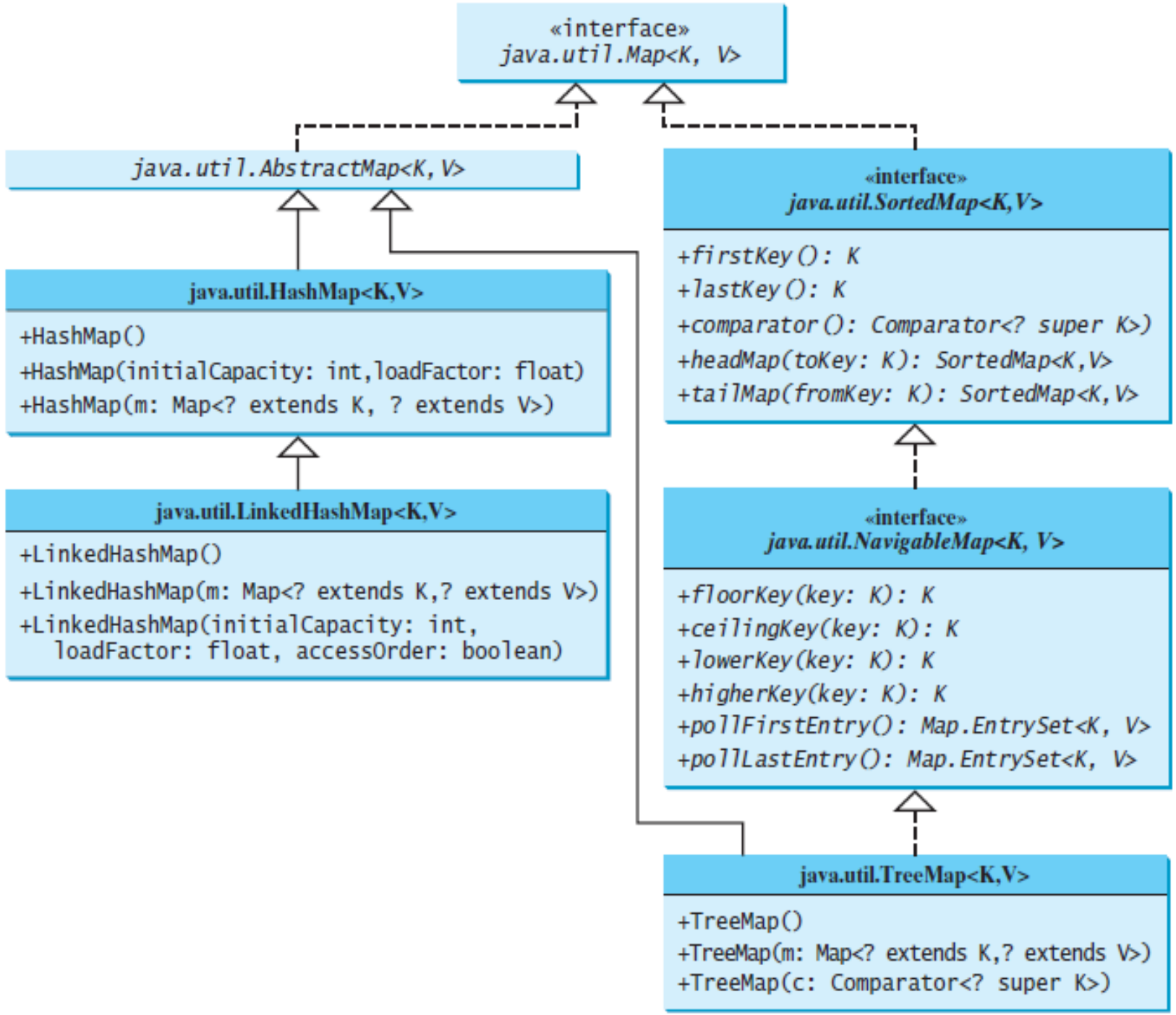
Map interface

- The AbstractMap class is a convenience class that implements all the methods in the Map interface except the entrySet() method.

- The SortedMap interface extends the Map interface to maintain the mapping in ascending order of keys with additional methods firstKey() and lastKey() for returning the lowest and highest key, headMap(toKey) for returning the portion of the map whose keys are less than toKey, and tailMap(fromKey) for returning the portion of the map whose keys are greater than or equal to fromKey.

- Hashtable implements the Map interface and is used in the same way as HashMap, except that Hashtable is synchronized.[Hashtable in not in course]

- Demo Program[TestMap.java]


«interface»
java.util.Map<K, V>
java.util.AbstractMap<K, V>
«interface»
java.util.SortedMap<K,V>
+firstKey(): K
+lastKey(): K
+comparator(): Comparator<? super K>)
+headMap(toKey: K): SortedMap<K,V>
+tailMap(fromKey: K): SortedMap<K,V>
java.util.HashMap<K,V>
+HashMap()
+HashMap(initialCapacity: int,loadFactor: float)
+HashMap(m: Map<? extends K, ? extends V>)
«interface»
java.util.NavigableMap<K, V>
+floorKey(key: K): K
+ceilingKey(key: K): K
+lowerKey(key: K): K
+higherKey(key: K): K
+pollFirstEntry(): Map.EntrySet<K, V>
+pollLastEntry(): Map.EntrySet<K, V>
java.util.LinkedHashMap<K,V>
+LinkedHashMap()
+LinkedHashMap(m: Map<? extends K,? extends V>)
+LinkedHashMap(initialCapacity: int,
loadFactor: float, accessOrder: boolean)
java.util.TreeMap<K,V>
+TreeMap()
+TreeMap(m: Map<? extends K,? extends V>)
+TreeMap(c: Comparator<? super K>)

```
// Demonstrate ArrayList.
import java.util.*;
class ArrayListDemo {
public static void main(String args[]) {
// Create an array list.
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " +
al.size());
// Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " +
al.size());
// Display the array list.
System.out.println("Contents of al: " + al);
// Remove elements from the array list.
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " +
al.size());
System.out.println("Contents of al: " + al);
}
}
```

The output from this program is shown here:

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

Convert an ArrayList into an array

```
import java.util.*;
class ArrayListToArray {
public static void main(String args[]) {
// Create an array list.
ArrayList<Integer> al = new ArrayList<Integer>();
// Add elements to the array list.
al.add(1);
al.add(2);
al.add(3);
al.add(4);
System.out.println("Contents of al: " + al);
// Get the array.
Integer ia[] = new Integer[al.size()];
ia = al.toArray(ia);
int sum = 0;
// Sum the array.
for(int i : ia) sum += i;
System.out.println("Sum is: " + sum);
}
}
```

The output from the program is shown here:

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

Demonstrate LinkedList

```
class LinkedListDemo {
public static void main(String args[]) {
// Create a linked list.
LinkedList<String> ll = new LinkedList<String>();
// Add elements to the linked list.
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
// Remove elements from the linked list.
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "+ ll);
// Remove first and last elements.
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "+ ll);
// Get and set a value.
String val = ll.get(2);
ll.set(2, val + " Changed");
System.out.println("ll after change: " + ll);
}
}
```

The output from this program is shown here:

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

Demonstrate HashSet.

```
import java.util.*;
class HashSetDemo {
public static void main(String args[]) {
// Create a hash set.
HashSet<String> hs = new HashSet<String>();
// Add elements to the hash set.
hs.add("B");
hs.add("A");
hs.add("D");
hs.add("E");
hs.add("C");
hs.add("F");
System.out.println(hs);
}
}
```

The following is the output from this program:

[D, A, F, C, B, E]

Demonstrate TreeSet

```
import java.util.*;
class TreeSetDemo {
public static void main(String args[]) {
// Create a tree set.
TreeSet<String> ts = new TreeSet<String>();
// Add elements to the tree set.
ts.add("C");
ts.add("A");
ts.add("B");
ts.add("E");
ts.add("F");
ts.add("D");
System.out.println(ts);
}
}
```

The output from this program is shown here:

[A, B, C, D, E, F]

Demonstrate ArrayDeque

```
import java.util.*;
class ArrayDequeDemo {
public static void main(String args[]) {
// Create a tree set.
ArrayDeque<String> adq = new
ArrayDeque<String>();
// Use an ArrayDeque like a stack.
adq.push("A");
adq.push("B");
adq.push("D");
adq.push("E");
adq.push("F");
System.out.print("Popping the stack: ");
while(adq.peek() != null)
System.out.print(adq.pop() + " ");
System.out.println();
}
}
```

The output is shown here:

```
Popping the stack: F E D B A
```

Demonstrate iterators

```
import java.util.*;
class IteratorDemo {
public static void main(String args[]) {
// Create an array list.
ArrayList<String> al = new ArrayList<String>();
// Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
// Use iterator to display contents of al.
System.out.print("Original contents of al: ");
Iterator<String> itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
// Modify objects being iterated.
ListIterator<String> litr = al.listIterator();
while(litr.hasNext()) {
String element = litr.next();
litr.set(element + "+");
}
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
// Now, display the list backwards.
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
String element = litr.previous();
System.out.print(element + " ");
}
System.out.println();
}
}
```

The output is shown here:
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+

Use the for-each for loop to cycle through a collection

```
import java.util.*;
class ForEachDemo {
public static void main(String args[]) {
// Create an array list for integers.
ArrayList<Integer> vals = new ArrayList<Integer>();
// Add values to the array list.
vals.add(1);
vals.add(2);
vals.add(3);
vals.add(4);
vals.add(5);
// Use for loop to display the values.
System.out.print("Original contents of vals: ");
for(int v : vals)
System.out.print(v + " ");
System.out.println();
// Now, sum the values by using a for loop.
int sum = 0;
for(int v : vals)
sum += v;
System.out.println("Sum of values: " + sum);
}
}
```

The output from the program is shown here:

```
Original contents of vals: 1 2 3 4 5
Sum of values: 15
```

Storing User-Defined Classes in Collections

```
import java.util.*;
class Address {
private String name;
private String street;
private String city;
private String state;
private String code;
Address(String n, String s, String c,
String st, String cd) {
name = n;
street = s;
city = c;
state = st;
code = cd;
}
public String toString() {
return name + "\n" + street + "\n" +
city + " " + state + " " + code; }
}
```

```
class MailList {
public static void main(String args[]) {
LinkedList<Address> ml = new LinkedList<Address>();
// Add elements to the linked list.
ml.add(new Address("J.W. West", "11 Oak Ave",
"Urbana", "IL", "61801"));
ml.add(new Address("Ralph Baker", "1142 Maple Lane",
"Mahomet", "IL", "61853"));
ml.add(new Address("Tom Carlton", "867 Elm St",
"Champaign", "IL", "61820"));
// Display the mailing list.
for(Address element : ml)
System.out.println(element + "\n");
System.out.println();
}
}
```

The HashMap Class

```
import java.util.*;
class HashMapDemo {
public static void main(String args[]) {
// Create a hash map.
HashMap<String, Double> hm = new HashMap<String, Double>();
// Put elements to the map
hm.put("John Doe", new Double(3434.34));
hm.put("Tom Smith", new Double(123.22));
hm.put("Jane Baker", new Double(1378.00));
hm.put("Tod Hall", new Double(99.22));
hm.put("Ralph Smith", new Double(-19.08));
// Get a set of the entries.
Set<Map.Entry<String, Double>> set = hm.entrySet();
// Display the set.
for(Map.Entry<String, Double> me : set) {
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());
}
System.out.println();
// Deposit 1000 into John Doe's account.
double balance = hm.get("John Doe");
hm.put("John Doe", balance + 1000);
System.out.println("John Doe's new balance: " +
hm.get("John Doe"));
}
}
```

Output from this program is shown here (the precise order may vary):

Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0
John Doe’s new balance: 4434.34

The TreeMap Class

```
import java.util.*;
class TreeMapDemo {
public static void main(String args[]) {
// Create a tree map.
TreeMap<String, Double> tm = new
TreeMap<String, Double>();
// Put elements to the map.
tm.put("John Doe", new Double(3434.34));
tm.put("Tom Smith", new Double(123.22));
tm.put("Jane Baker", new
Double(1378.00));
tm.put("Tod Hall", new Double(99.22));
tm.put("Ralph Smith", new Double(-19.08));
// Get a set of the entries.
Set<Map.Entry<String, Double>> set =
tm.entrySet();
// Display the elements.
for(Map.Entry<String, Double> me : set) {
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());
}
System.out.println();
// Deposit 1000 into John Doe's account.
double balance = tm.get("John Doe");
tm.put("John Doe", balance + 1000);
System.out.println("John Doe's new balance:
" +
tm.get("John Doe"));
}
}
```

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22
John Doe's current balance:
4434.34
```

Using a Comparator

```
import java.util.*;
// A reverse comparator for strings.
class MyComp implements Comparator<String> {
public int compare(String a, String b) {
String aStr, bStr;
aStr = a;
bStr = b;
// Reverse the comparison.
return bStr.compareTo(aStr);
}
// No need to override equals.
}
class CompDemo {
public static void main(String args[]) {
// Create a tree set.
TreeSet<String> ts = new TreeSet<String>(new
MyComp());
// Add elements to the tree set.
ts.add("C");
ts.add("A");
ts.add("B");
ts.add("E");
ts.add("F");
ts.add("D");
// Display the elements.
for(String element : ts)
System.out.print(element + " ");
System.out.println();
}
}
```

As the following output shows, the tree is now stored in reverse order:

```
F E D C B A
```

Use a comparator to sort accounts by last name.

```
import java.util.*;
// Compare last whole words in two strings.
class TComp implements Comparator<String> {
public int compare(String a, String b) {
int i, j, k;
String aStr, bStr;
aStr = a;
bStr = b;
// Find index of beginning of last name.
i = aStr.lastIndexOf(' ');
j = bStr.lastIndexOf(' ');
k = aStr.substring(i).compareTo(bStr.substring(j));
if(k==0) // last names match, check entire name
return aStr.compareTo(bStr);
else
return k;
}
// No need to override equals.
}
```

```
class TreeMapDemo2 {
public static void main(String args[]) {
// Create a tree map.
TreeMap<String, Double> tm = new TreeMap<String, Double>(new TComp());
// Put elements to the map.
tm.put("John Doe", new Double(3434.34));
tm.put("Tom Smith", new Double(123.22));
tm.put("Jane Baker", new Double(1378.00));
tm.put("Tod Hall", new Double(99.22));
tm.put("Ralph Smith", new Double(-19.08));
// Get a set of the entries.
Set<Map.Entry<String, Double>> set = tm.entrySet();
// Display the elements.
for(Map.Entry<String, Double> me : set) {
System.out.print(me.getKey() + ": ");
System.out.println(me.getValue());
}
System.out.println();
// Deposit 1000 into John Doe's account.
double balance = tm.get("John Doe");
tm.put("John Doe", balance + 1000);
System.out.println("John Doe's new balance: " +
tm.get("John Doe"));
}
}
```

Jane Baker: 1378.0
John Doe: 3434.34
Todd Hall: 99.22
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe’s new balance: 4434.34

//Different Algorithms

```
import java.util.*;
class AlgorithmsDemo {
public static void main(String args[]) {
// Create and initialize linked list.
LinkedList<Integer> ll = new LinkedList<Integer>();
ll.add(-8);
ll.add(20);
ll.add(-20);
ll.add(8);
// Create a reverse order comparator.
Comparator<Integer> r = Collections.reverseOrder();
// Sort list by using the comparator.
Collections.sort(ll, r);
System.out.print("List sorted in reverse: ");
for(int i : ll)
System.out.print(i+ " ");
System.out.println();
// Shuffle list.
Collections.shuffle(ll);
// Display randomized list.
System.out.print("List shuffled: ");
for(int i : ll)
System.out.print(i + " ");
System.out.println();
System.out.println("Minimum: " +
Collections.min(ll));
System.out.println("Maximum: " +
Collections.max(ll));
}
}
```

Output from this program is shown here:

```
List sorted in reverse: 20 8 -8 -20
List shuffled: 20 -20 8 -8
Minimum: -20
Maximum: 20
```

Demonstrate Arrays

```
import java.util.*;
class ArraysDemo {
public static void main(String args[]) {
// Allocate and initialize array.
int array[] = new int[10];
for(int i = 0; i < 10; i++)
array[i] = -3 * i;
// Display, sort, and display the array.
System.out.print("Original contents: ");
display(array);
Arrays.sort(array);
System.out.print("Sorted: ");
display(array);
// Fill and display the array.
Arrays.fill(array, 2, 6, -1);
System.out.print("After fill(): ");
display(array);
// Sort and display the array.
Arrays.sort(array);
System.out.print("After sorting again: ");
display(array);
// Binary search for -9.
System.out.print("The value -9 is at location ");
int index =
Arrays.binarySearch(array, -9);
System.out.println(index);
}
static void display(int array[]) {
for(int i: array)
System.out.print(i + " ");
System.out.println();}}
```

The following is the output from this program:

```
Original contents: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Sorted: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
After fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
After sorting again: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
The value -9 is at location 2
```

Demonstrate various Vector operations

```
import java.util.*;
class VectorDemo {
public static void main(String args[]) {
// initial size is 3, increment is 2
Vector<Integer> v = new Vector<Integer>(3, 2);
System.out.println("Initial size: " + v.size());
System.out.println("Initial capacity: " +
v.capacity());
v.addElement(1);
v.addElement(2);
v.addElement(3);
v.addElement(4);
System.out.println("Capacity after four additions: " +
v.capacity());
v.addElement(5);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(6);
v.addElement(7);
System.out.println("Current capacity: " +v.capacity());
v.addElement(9);
v.addElement(10);
System.out.println("Current capacity: " + v.capacity());
v.addElement(11);
v.addElement(12);
System.out.println("First element: " + v.firstElement());
System.out.println("Last element: " + v.lastElement());
if(v.contains(3))
System.out.println("Vector contains 3.");
// Enumerate the elements in the vector.
Enumeration vEnum = v.elements();
System.out.println("\nElements in vector:");
while(vEnum.hasMoreElements())
System.out.print(vEnum.nextElement() + " ");
System.out.println();
}
}
```

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
```

Demonstrate the Stack class

```
import java.util.*;
class StackDemo {
static void showpush(Stack<Integer> st, int a) {
st.push(a);
System.out.println("push(" + a + ")");
System.out.println("stack: " + st);
}
static void showpop(Stack<Integer> st) {
System.out.print("pop -> ");
Integer a = st.pop();
System.out.println(a);
System.out.println("stack: " + st);
}
public static void main(String args[]) {
Stack<Integer> st = new Stack<Integer>();
System.out.println("stack: " + st);
showpush(st, 42);
showpush(st, 66);
showpush(st, 99);
showpop(st);
showpop(st);
showpop(st);
try {
showpop(st);
} catch (EmptyStackException e) {
System.out.println("empty stack");
}
}
}
```

Demonstrate Calendar

```
import java.util.Calendar;
class CalendarDemo {
public static void main(String args[]) {
String months[] = {
"Jan", "Feb", "Mar", "Apr",
"May", "Jun", "Jul", "Aug",
"Sep", "Oct", "Nov", "Dec"};
// Create a calendar initialized with the
// current date and time in the default
// locale and timezone.
Calendar calendar = Calendar.getInstance();
// Display current time and date information.
System.out.print("Date: ");
System.out.print(months[calendar.get(Calendar.MONTH)]);
System.out.print(" " + calendar.get(Calendar.DATE) + " ");
System.out.println(calendar.get(Calendar.YEAR));
System.out.print("Time: ");
System.out.print(calendar.get(Calendar.HOUR) + ":");
System.out.print(calendar.get(Calendar.MINUTE) + ":");
System.out.println(calendar.get(Calendar.SECOND));
// Set the time and date information and display it.
calendar.set(Calendar.HOUR, 10);
calendar.set(Calendar.MINUTE, 29);
calendar.set(Calendar.SECOND, 22);
System.out.print("Updated time: ");
System.out.print(calendar.get(Calendar.HOUR) + ":");
System.out.print(calendar.get(Calendar.MINUTE) + ":");
System.out.println(calendar.get(Calendar.SECOND));
}
}
```

GregorianCalendar

```
import java.util.*;
class GregorianCalendarDemo {
public static void main(String args[]) {
String months[] = {
"Jan", "Feb", "Mar", "Apr",
"May", "Jun", "Jul", "Aug",
"Sep", "Oct", "Nov", "Dec"};
int year;
// Create a Gregorian calendar initialized
// with the current date and time in the
// default locale and timezone.
GregorianCalendar gcalendar = new GregorianCalendar();
// Display current time and date information.
System.out.print("Date: ");
System.out.print(months[gcalendar.get(Calendar.MONTH)]);
System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
System.out.println(year = gcalendar.get(Calendar.YEAR));
System.out.print("Time: ");
System.out.print(gcalendar.get(Calendar.HOUR) + ":");
System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
System.out.println(gcalendar.get(Calendar.SECOND));
// Test if the current year is a leap year
if(gcalendar.isLeapYear(year)) {
System.out.println("The current year is a leap year");
}
else {
System.out.println("The current year is not a leap year");
}
}
}
```

Demonstrate Timer and TimerTask

```
import java.util.*;
class MyTimerTask extends TimerTask
{
public void run() {
System.out.println("Timer task
executed.");
}
}
class TTest {
public static void main(String args[]) {
MyTimerTask myTask = new MyTimerTask();
Timer myTimer = new Timer();
/* Set an initial delay of 1 second,
then repeat every half second.
*/
myTimer.schedule(myTask, 1000,
500);
try {
Thread.sleep(5000);
} catch (InterruptedException exc) {}
myTimer.cancel();
}
}
```

Currency

```
import java.util.*;
class CurDemo {
public static void main(String args[]) {
Currency c;
c = Currency.getInstance(Locale.US);
System.out.println("Symbol: " + c.getSymbol());
System.out.println("Default fractional digits: " +
c.getDefaultFractionDigits());
}
}
```

The output is shown here:

```
Symbol: $
Default fractional digits: 2
```